



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Silo: A General Purpose API and Scientific Database

M. C. Miller

March 27, 2014

High Performance Parallel I/O

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Chapter 20

Silo: A General Purpose API and Scientific Database

Mark Miller

Lawrence Livermore National Laboratory

20.1	Canonical Use Case: ALE3D Restart and VisIt Visualization Workflow	236
20.2	Software, Hardware, and Performance	237
20.3	MIF and SSF Scalable I/O Paradigms	239
20.4	Successes with HDF5 as Middleware	241
20.5	Conclusion	242
	Bibliography	244

This chapter presents results and lessons learned in developing and using Silo [?]. Silo is neither an application nor an application-specific I/O framework or library. Instead, Silo is a general purpose application programming interface (API) and I/O library for reading and writing a variety of scientific computing data objects to HDF5 files.

Silo supports several mesh types including gridless, structured, unstructured, arbitrary, adaptive (AMR), and solid model (CSG) meshes. It supports piecewise-constant and piecewise-linear variables (e.g., fields) defined on the node, edge, face or volume elements of such meshes as well as the decomposition of meshes into subsets including boundaries, materials, and part assemblies.

Although Silo is a serial library, key features enable it to be applied effectively and scalably in parallel.

There is no specific science application to motivate Silo's development or use. Silo was developed to address a fundamental software engineering challenge. That is, to spur the development of common tools by enabling storage, sharing and exchange of data among diverse scientific computing applications through a common API and database.

This chapter illustrates the use of Silo by describing a canonical use case. The chapter will use as examples the ALE3D multi-physics simulation, the VisIt visualization application, the Lustre file system, and the way Silo is used to store and exchange data between these components.

20.1 Canonical Use Case: ALE3D Restart and VisIt Visualization Workflow

ALE3D [?] is a multi-physics application utilizing arbitrary Lagrangian-Eulerian (ALE) techniques. It models fluid and elastic-plastic response on unstructured grids of hexahedra. ALE3D integrates a variety of multi-physics capabilities through an operator-splitting approach including heat conduction, chemical kinetics with species diffusion, incompressible flow and a wide range of material and chemistry models. Fully featured ALE3D simulations have been run with good scaling behavior up to as many as 128,000 MPI [1] tasks.

ALE3D uses Silo to produce restart files, plot files, time-history files, and mesh-to-mesh linking files. Restart files are used to restart ALE3D after execution has been terminated. Plot files are used in visualization and analysis. Time-history files capture high time resolution data at specific user-defined points. Mesh-to-mesh linking files are produced periodically when workflows require ALE3D solution data to be integrated and exchanged with applications modeling other physical phenomena.

For restart and plot files, the key data objects ALE3D writes to the files are the main mesh, the material composition of the main mesh, and key physical variables (e.g., fields) defined on the main mesh such as pressure, velocity, mass, flux, etc.

The main mesh is an unstructured cell data (UCD) mesh; an arbitrary arrangement of connected 3D hexahedral mesh elements (see Figure 20.1). For computation in a distributed, scalable parallel setting, the main mesh is decomposed into pieces called domains ranging in size from 2.5 to 25 and typically 10 thousand mesh elements. To reduce communication in parallel computation, neighboring domains typically contain copies of each other's elements along their shared boundaries. In Silo parlance, these copies are called ghost elements.

The total number elements in the main mesh is a commonly used metric for the scale or size of the problem being simulated. As problem size increases, typically the number of domains increases but the size of each domain (e.g., per-domain element count) remains roughly the same. When I/O performance and scaling is studied, the focus is on those Silo objects the application produces whose size varies with problem size.

Once decomposed, the main mesh is forevermore stored and exchanged via Silo in its decomposed state. All tools and applications used in Silo-enabled HPC workflows are designed to interact with the data in its decomposed state. In fact, all applications are designed so that any single MPI task can manage multiple domains simultaneously. For example, given a mesh decomposed into 60 domains, both ALE3D and VisIt can run one domain per task on 60 MPI tasks, 2:1 on 30 tasks, 3:1 on 20 tasks or even 3:1 on 12 together with 4:1

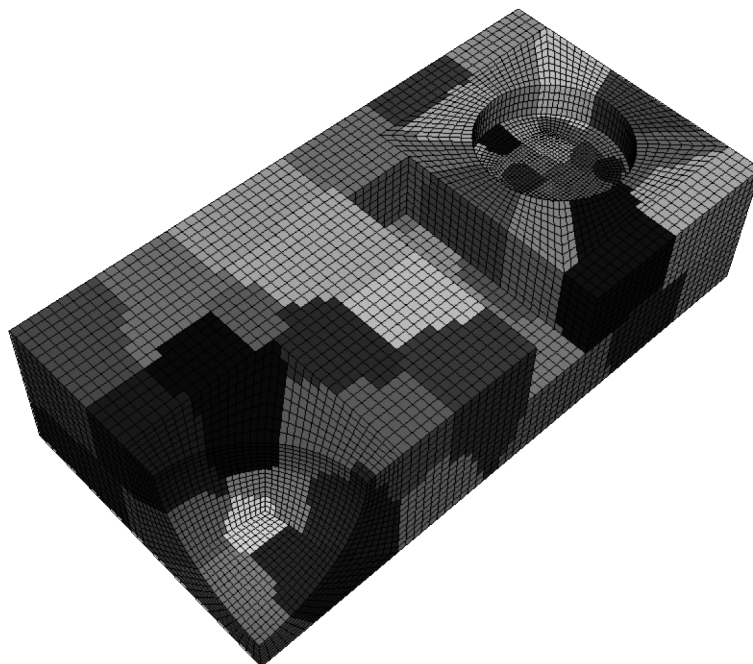


FIGURE 20.1: Example of an ALE3D mesh decomposed into domains. [Image courtesy of Bert Still (LLNL).]

on 6 of 18 total tasks. This gives applications great flexibility in allocating compute resources.

20.2 Software, Hardware, and Performance

In a typical HPC workflow, ALE3D will run for hours to days on tens of thousands of MPI tasks, producing restart files about once per hour and plot files perhaps several times per hour on the Lustre file system. Users will run VisIt on approximately one-tenth the compute allocation used for ALE3D to visualize and perform sanity checks as the simulation proceeds.

Silo is a serial I/O library. Therefore, when ALE3D writes a restart file, it first determines the number of Silo files to be used to store all the domains of the main mesh: call this number N . N is selected completely independently of the number of MPI tasks and domains. A good choice for N is to match

the number of independent I/O pathways available from compute nodes to the file system. Typically, this number is between 8 and 1024 depending on problem size and the compute and file system resources involved.

ALE3D groups MPI-tasks into N groups and each group is responsible for creating one of the N files. At any one moment, only one MPI task from each group has exclusive access to the file. Hence, I/O is serial within a group. However, because one task in each group is writing to its group's own file, simultaneously, I/O is parallel across groups. Within a group, access to the

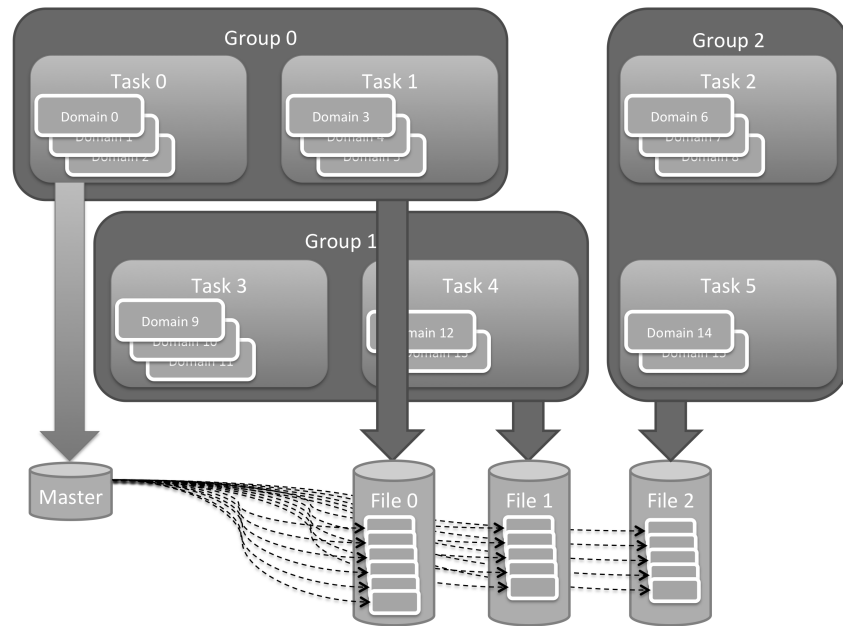


FIGURE 20.2: Example Silo-based MIF-I/O. There are 6 MPI tasks. Tasks 0–3 have 3 domains. Tasks 4 and 5 have 2 domains. Tasks are divided into 3 I/O groups. Task 0 creates file 0 and writes its 3 domains in 3 sub-directories in the file. Simultaneously, tasks 1 and 2 create files 1 and 2 and write their domains. There are 3 parallel streams of data from the application.

group's file is handled in a round-robin fashion. The first MPI task in the group creates the file and then iterates over all domains it has. For each domain, it creates a sub-directory within the file (e.g., a separate namespace for Silo objects) and writes all the Silo objects (the main mesh domain, the material composition of the domain, the mesh variables defined on the domain) to that directory. It repeats this process for each domain. Then, the first MPI task closes the Silo file and hands off exclusive access to the next task in the group. That MPI task opens the file and iterates over all domains in the same way.

Exclusive access to the file is then handed off to the next task. This process, shown in Figure 20.2, continues until all processors in the group have written their domains to unique sub-directories in the file.

After all groups have finished writing their Silo files, a final step involves creating a master Silo file which contains special Silo objects (called multi-block objects) that point at all the pieces of mesh (domains) scattered about in the N files.

Setting N to be equal to the number of MPI tasks, results in a file-per-process configuration, which is typically not recommended for users. However, some applications do indeed choose to run this way with good results. Alternatively, setting N equal to 1 results in effectively serializing the I/O and is certainly not recommended. For large, parallel runs, there is a sweet spot in the selection of N which results in peak I/O performance rates. If N is too large, the I/O subsystem will likely be overwhelmed; setting it too small will likely underutilize the system resources. This is illustrated in Figure 20.3 for different numbers of files and MPI task counts.

20.3 MIF and SSF Scalable I/O Paradigms

This approach to using Silo for scalable, parallel I/O was originally developed in the late 1990s by Rob Neely, a lead software architect on ALE3D at the time. This approach is sometimes called “Poor Man’s Parallel I/O.” It and variations thereof have since been adopted and used productively through several transitions in orders of magnitude of MPI task counts from hundreds then to hundreds of thousands today.

During this same period of time, R&D efforts in scalable, parallel I/O largely focused on the case of concurrent I/O to a single, shared file. There has long been the belief that concurrent I/O to a single shared file is the best way to achieve truly scalable I/O. By contrast, these approaches are sometimes called “Rich Man’s Parallel I/O” because the underlying software subsystems support parallel I/O to a single shared file, natively.

The key distinction between the Poor Man’s and Rich Man’s approaches is the need for applications to manage distribution of data among multiple files. For this reason, it is technically appropriate to refer to these two approaches as Multiple Independent File (MIF) and Single Shared File (SSF) parallel I/O.

There are a large number of advantages to MIF-IO over SSF-IO.

1. MIF-IO is a much simpler programming model because it frees developers from having to think in terms of collective I/O operations. The code to write data from one MPI task doesn’t depend on or involve other tasks. For large multi-physics applications where the size, shape

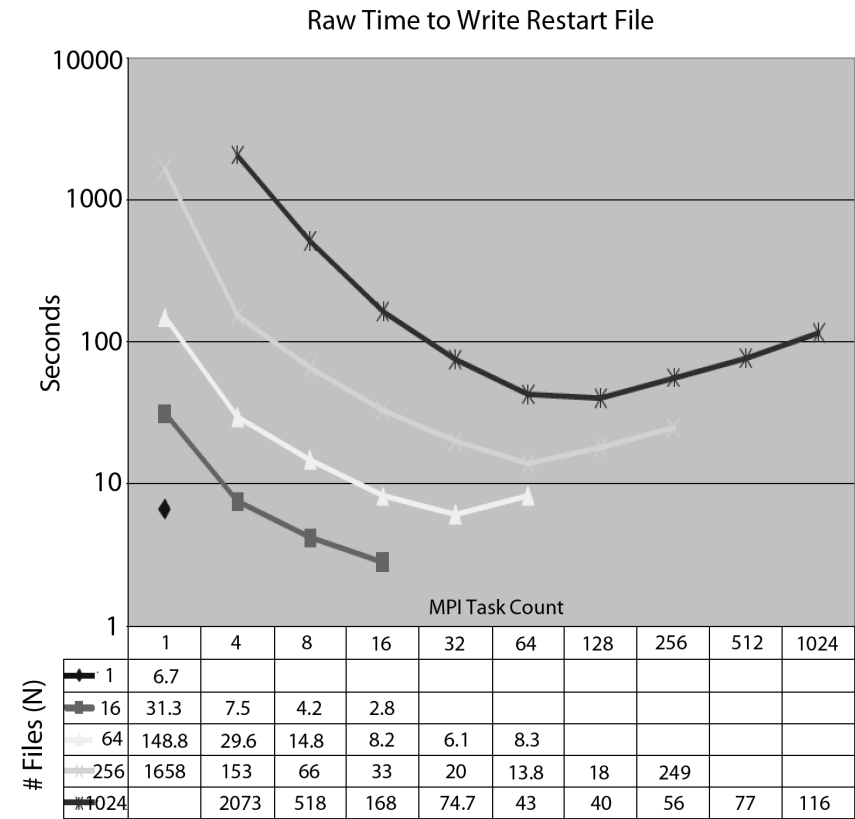


FIGURE 20.3: Variation in ALE3D I/O performance as number of file parts is varied. [Image courtesy of Rob Neely (LLNL).]

and even existence of data can vary dramatically among MPI tasks, this is invaluable in simplifying I/O code development.

- 2. MIF-IO alleviates any need for global-to-local and local-to-global remapping upon every exchange of data between the application and its file.
- 3. In-transit compression is easier to apply in a MIF-IO setting because processors are freed from having to coordinate changes in data sizing as it is moved to the file.
- 4. For MIF-IO, good performance demands very little in the way of extra/advanced features from the underlying I/O hardware and file system. A relatively dumb file system can get it right and perform well.

5. Application controlled throttling of I/O is easily supported in a MIF-IO setting because the number of concurrent operations is explicitly controlled. This can help to avoid overloading the underlying I/O subsystems.
6. MIF-IO is consistent with the way leading-edge commercial “big data” I/O in map-reduce operations is handled. Data sets are broken into pieces and stored in the file system as a collection of shards and different numbers of parallel tasks can process different numbers of shards.

With minor variations, the process by which ALE3D writes plot files is completely analogous to restart files. However, the application will often enable some “in-transit” transformations on data destined for plot files that it would otherwise not perform for restart files. This can include transformations from double to single precision, the addition of ghost zone layers enveloping domains, the inclusion of domain spatial and variable extents as well as both lossless and lossy compression. In earlier systems where computations were performed on Crays and visualizations on SGIs, Silo would also perform transformations from Cray to SGI floating point representations.

Nonetheless, both restart and plot files, because they are Silo files, are still visualizable by VisIt. Plot files are designed to be a little more convenient and optimized for visualization in VisIt while restart files are optimized for restarting ALE3D.

When VisIt [?] reads a Silo restart or plot file, it starts by opening the master file. From that file, VisIt’s metadata server determines all of the objects in the file, their names and types and then populates VisIt’s GUI menus based upon what it finds in the files. There are a number of interesting aspects to how VisIt manages data distributed across multiple files. What is most relevant here is that VisIt essentially piggy-backs its parallel processing paradigm on top of the parallel decomposition already provided by the data producer.

Like ALE3D, VisIt can process multiple domains on each MPI task. When the user starts VisIt, she specifies a suitable number of MPI tasks to use. VisIt then automatically assigns domains to tasks (it has a number of different algorithms for doing this depending on various factors). Each task then uses information from the master file to determine which file(s) contain the domains it needs and then opens those files and reads the relevant domains from them.

If the data producer bothered to write per-domain spatial and variable extents, VisIt can use such information to accelerate various operations. For example, to display a slice of a mesh, VisIt can exclude from processing any domain whose spatial extents do not intersect the slice. Likewise, when displaying iso-surfaces, VisIt can exclude from consideration any domain whose variable extents do not contain the iso-values of interest.

20.4 Successes with HDF5 as Middleware

Silo is built on top of and uses the HDF5 library. Because of this, it is possible to alter the manner in which Silo uses HDF5 without having to alter applications above it. We have used this to advantage on numerous occasions to address various file system performance and reliability issues. Examples of some of those capabilities are briefly described here.

1. Application-level checksumming: To address file system reliability issues a checksumming capability in HDF5 was enabled. As Silo applications write data, the data is checksummed by HDF5 in-transit to the file. No attempt to detect errors during write is made. However, checksums are also computed and compared upon read. In the event of a checksum error on read, the application may attempt to re-read the data. However, repeated read failures generally indicate an error most likely occurred during write and the data is corrupted.
2. Mesh-aware compression: To experiment with advanced compression techniques that can take advantage of mesh structure to optimize data locality during compression, we added the compression algorithms as HDF5 filters. This enables the compression to be applied in-transit as the data is written from the Silo application. We have used these compression methods to improve I/O performance and reduce file sizes for certain cases by as much as $5\times$.
3. Block-based Virtual File Driver (VFD): To address file system performance issues, a custom VFD designed to optimize I/O for BG/Q class systems. The VFD coalesces I/O requests from Silo applications and separates I/O for small objects (Silo metadata) from larger objects (application raw data). It breaks a file into large blocks consisting entirely of either metadata or raw data and keeps an application specified number of blocks cached in memory at any one time. An LRU algorithm is used to preempt cache blocks to disk when new blocks are needed. This VFD has demonstrated $30\text{--}50\times$ performance improvements on BG/Q systems. Results are illustrated in Figure 20.4.

20.5 Conclusion

Silo was developed to address a software engineering challenge; to enable the development of common tools with which applications can share

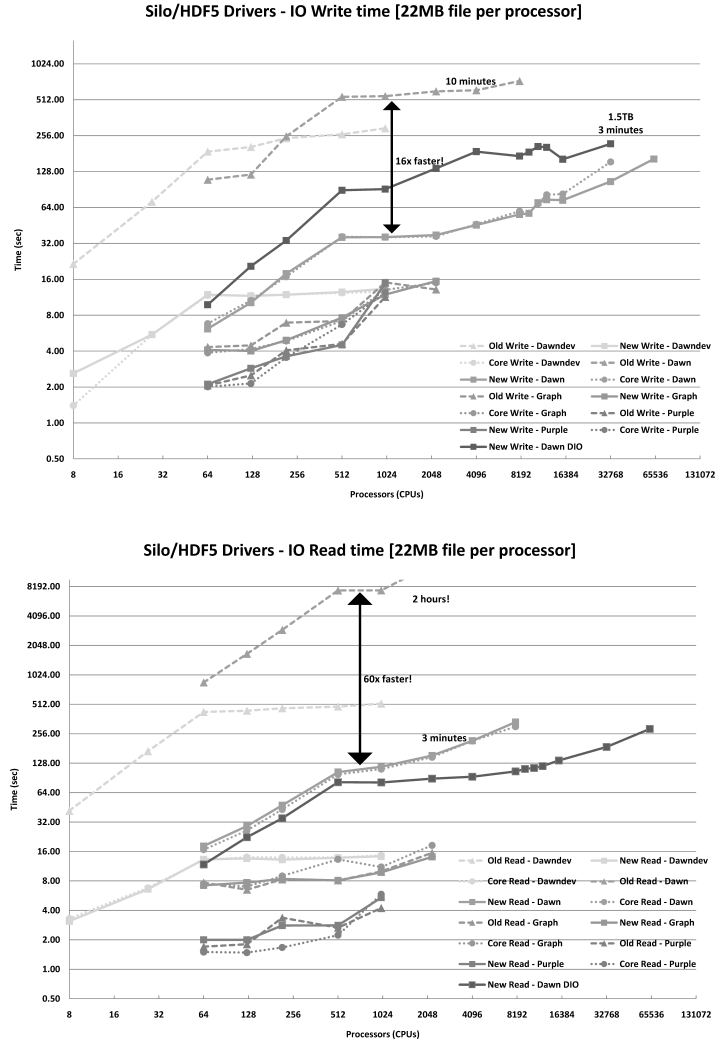


FIGURE 20.4: Comparison of MIF-IO Write and Read performance on 3 ASC Systems (Purple [?], Dawn [?] and Graph [?]) and Improvement in I/O performance with block-based VFD. [Image courtesy of Michael Collette (LLNL).]

and exchange diverse scientific datasets. As a serial library, Silo has nonetheless demonstrated scalable performance for many HPC applications (ALE3D, Kull, Overlink, PMesh, Ares and VisIt are some examples) using the MIF-IO parallel I/O approach. MIF-IO has demonstrated good performance to 100,000 tasks and is currently in process of being scaled to 1,000,000 cores. MIF-IO has a number of advantages over SSF-IO making its application very attrac-

tive in HPC workflows involving diverse and disparate datasets with dramatic variation in size, shape and existence of data across MPI tasks. Silo's success is due in large part to the performance and capabilities available in HDF5.

20.6 Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory in part under Contract W-7405-Eng-48 and in part under Contract DE-AC52-07NA27344. Document number LLNL-BOOK-652399.

Bibliography

- [1] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.